

Presentación para usar con el libro de texto **Diseño y aplicaciones de algoritmos**, de MT Goodrich y R. Tamassia, Wiley, 2015

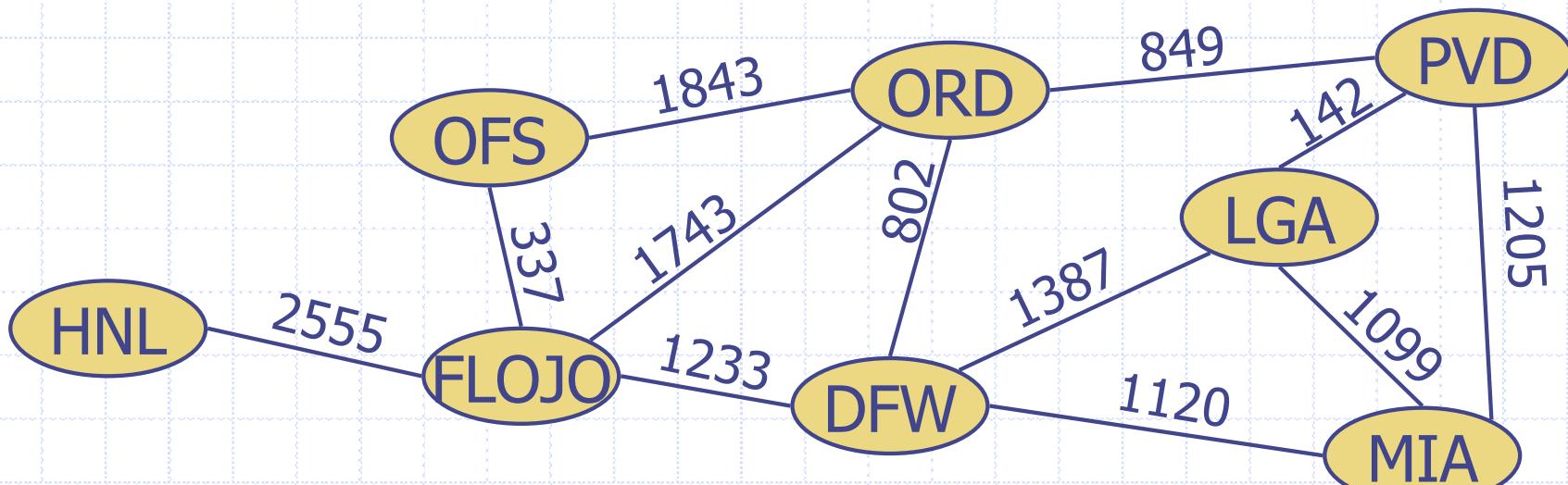
# Caminos más cortos



Lightning strike, 2009. U.S. government image. NOAA.

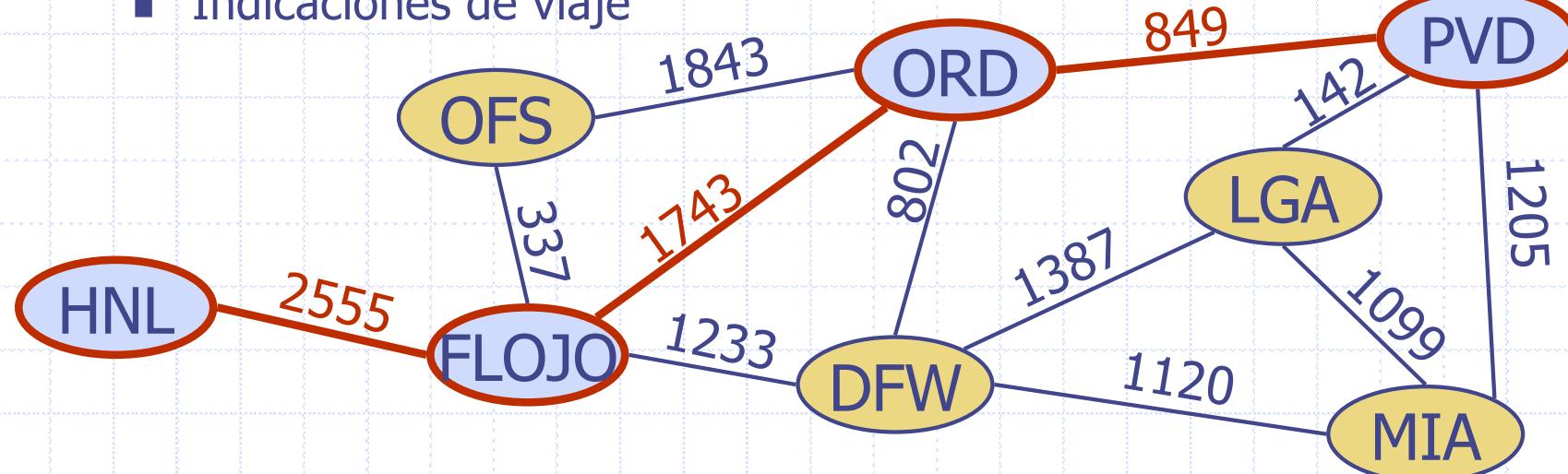
# Grafos ponderados (pesados)

- En un grafo ponderado (o pesado), cada arista tiene un valor numérico asociado, llamado peso de la arista.
- Los pesos de las aristas pueden representar distancias, costos, etc.
- Ejemplo:
  - En un grafo de ruta de vuelo, el peso de una arista representa la distancia en millas entre los aeropuertos de los puntos finales.



# Caminos más cortos

- Dado un grafo ponderado y dos vértices  $u$  y  $v$ , queremos encontrar un camino de peso total mínimo entre  $u$  y  $v$ .
  - La longitud de un camino es la suma de los pesos de sus aristas.
- Ejemplo:
  - El camino más corto entre Providence y Honolulu
- Aplicaciones
  - enrutamiento de paquetes de Internet
  - Reservas de vuelos
  - Indicaciones de viaje



# Propiedades del camino más corto

## Propiedad 1:

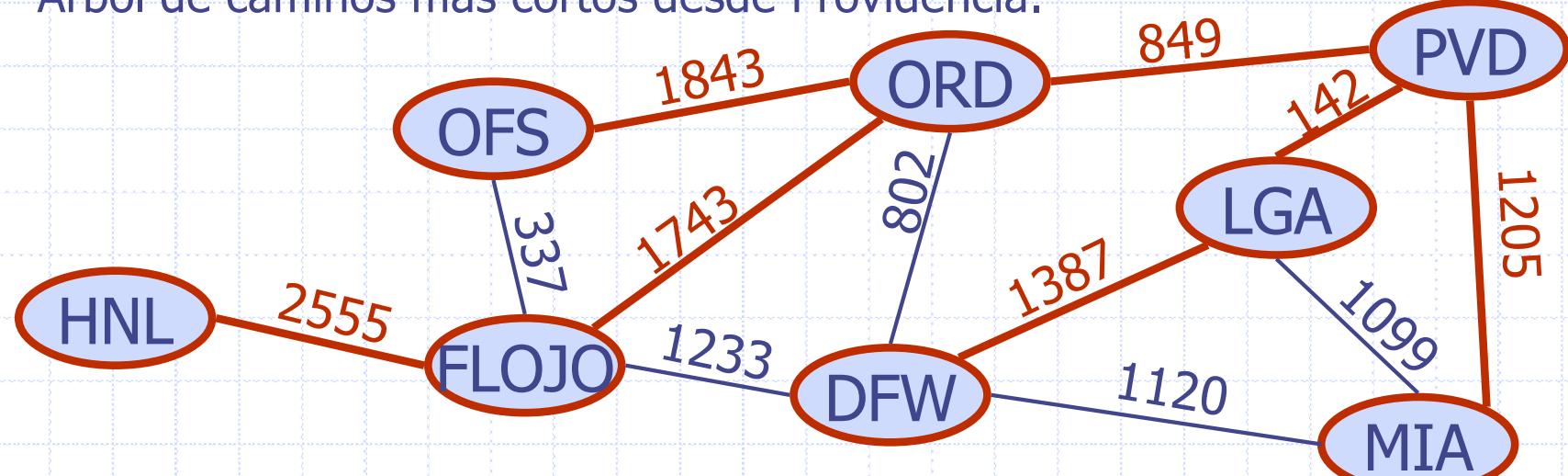
Un subtrayecto de un camino más corto es en sí mismo un camino más corto.

## Propiedad 2:

Hay un árbol de caminos más cortos desde un vértice inicial hasta todos los demás vértices.

## Ejemplo:

Árbol de caminos más cortos desde Providencia.



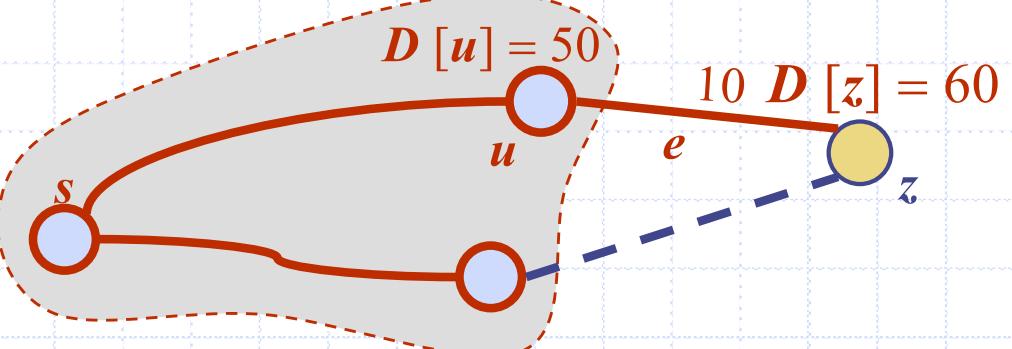
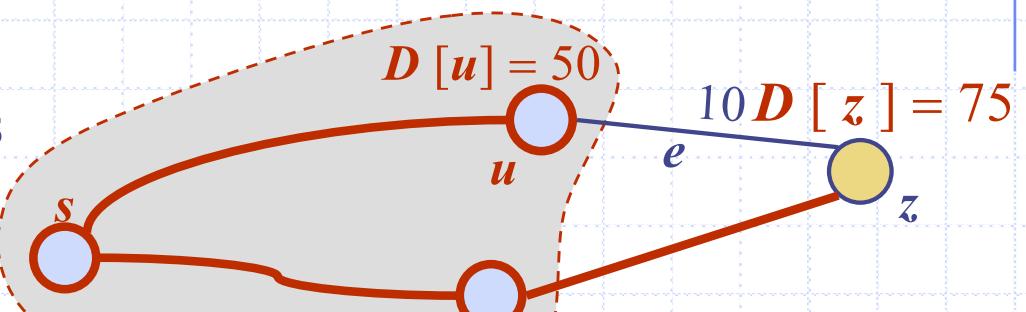
# Algoritmo de Dijkstra

- La distancia de un vértice  $v$  a un vértice  $s$  es la longitud del camino más corto entre  $s$  y  $v$
- El algoritmo de Dijkstra calcula las distancias de todos los vértices desde un vértice inicial determinado .
- Asumimos que:
  - el grafo es conectado
  - las aristas no son dirigidas
  - los pesos de las aristas **no** son negativos
- Creamos una " nube " de vértices, comenzando con  $s$  y eventualmente cubriendo todos los vértices.
- Almacenamos con cada vértice  $v$  una **etiqueta  $D[v]$**  que representa la distancia de  $v$  a  $s$  en el subgrafo que consiste en la nube y sus vértices adyacentes
- En cada paso
  - Agregamos a la nube el vértice  $u$  fuera de la nube con la etiqueta de distancia más pequeña,  $D[u]$
  - Actualizamos las etiquetas de los vértices adyacentes a  $u$ .

# Relajación de aristas

- Considere una arista  $e = (u, z)$  tal que
  - $u$  es el vértice agregado más recientemente a la nube
  - $z$  no está en la nube
- La relajación de la arista  $e$  actualiza la distancia  $d(z)$  de la siguiente manera:

$$D[z] \leftarrow \min \{D[z], D[u] + \text{peso}(e)\}$$



# Algoritmo de Dijkstra: detalles

**Algorithm** DijkstraShortestPaths( $G, v$ ):

**Input:** A simple undirected weighted graph  $G$  with nonnegative edge weights, and a distinguished vertex  $v$  of  $G$

**Output:** A label,  $D[u]$ , for each vertex  $u$  of  $G$ , such that  $D[u]$  is the distance from  $v$  to  $u$  in  $G$

$D[v] \leftarrow 0$

**for** each vertex  $u \neq v$  of  $G$  **do**

$D[u] \leftarrow +\infty$

Let a priority queue,  $Q$ , contain all the vertices of  $G$  using the  $D$  labels as keys.

**while**  $Q$  is not empty **do**

// pull a new vertex  $u$  into the cloud

$u \leftarrow Q.\text{removeMin}()$

**for** each vertex  $z$  adjacent to  $u$  such that  $z$  is in  $Q$  **do**

// perform the *relaxation* procedure on edge  $(u, z)$

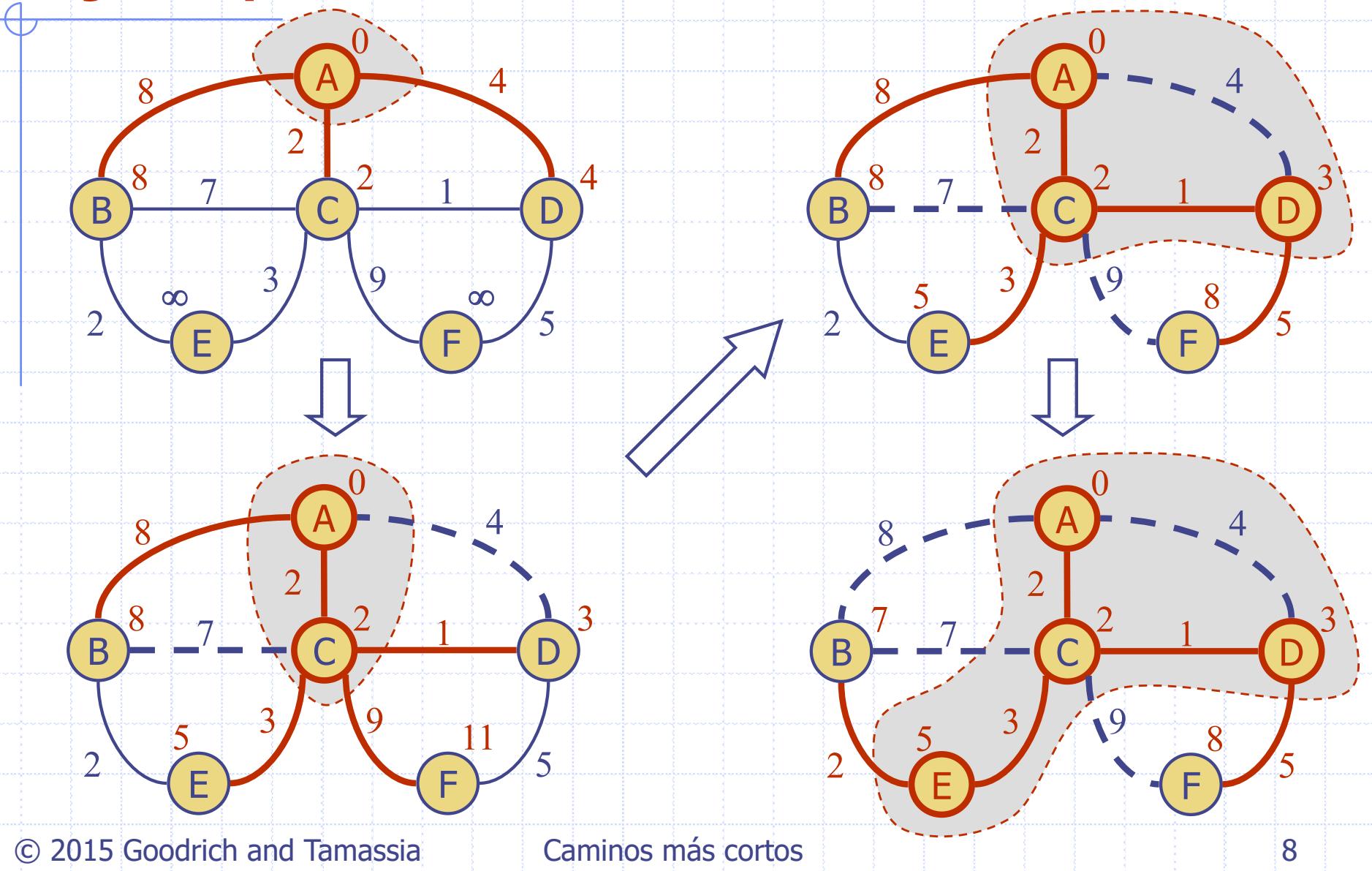
**if**  $D[u] + w((u, z)) < D[z]$  **then**

$D[z] \leftarrow D[u] + w((u, z))$

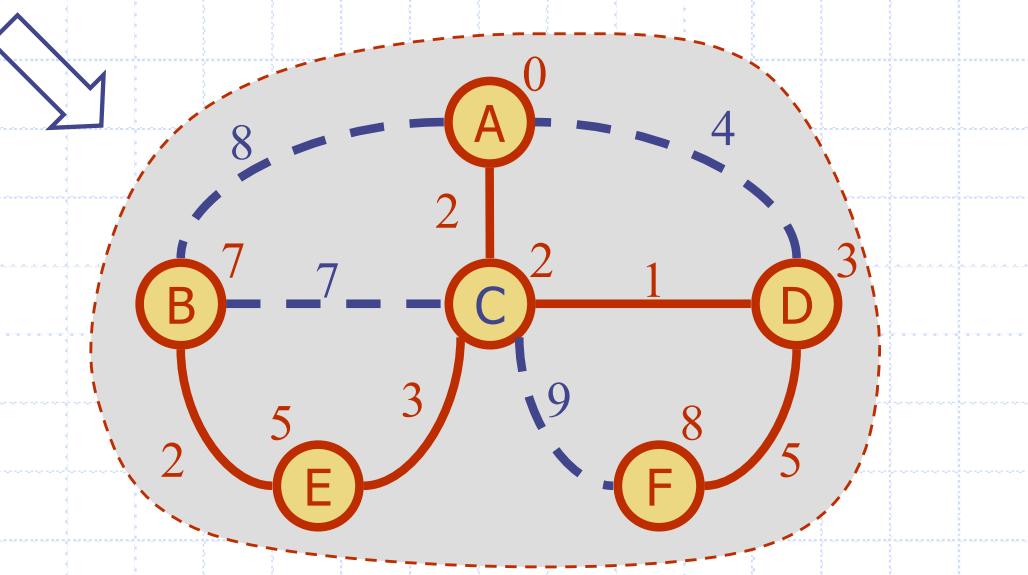
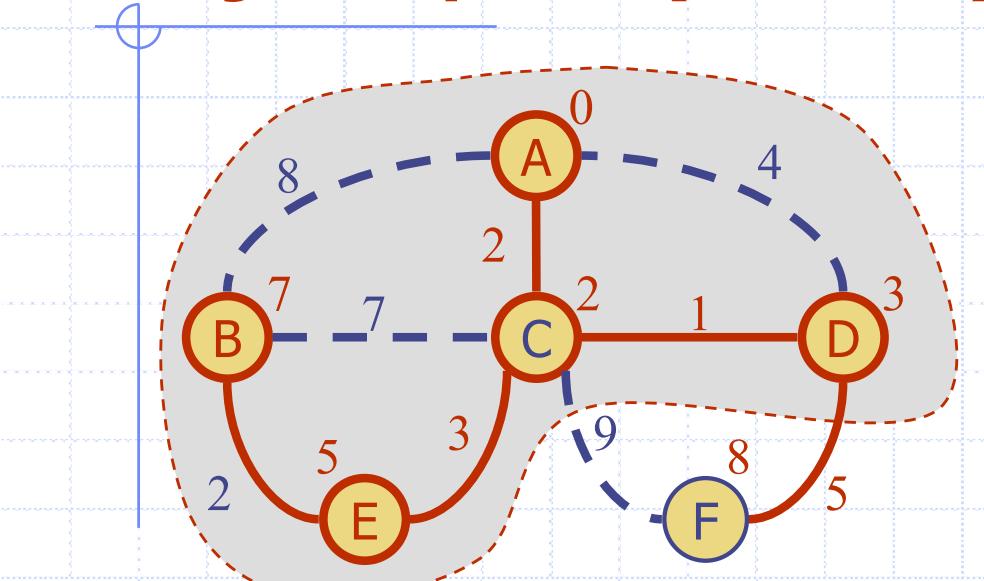
Change the key for vertex  $z$  in  $Q$  to  $D[z]$

**return** the label  $D[u]$  of each vertex  $u$

# Ejemplo



# Ejemplo (cont.)

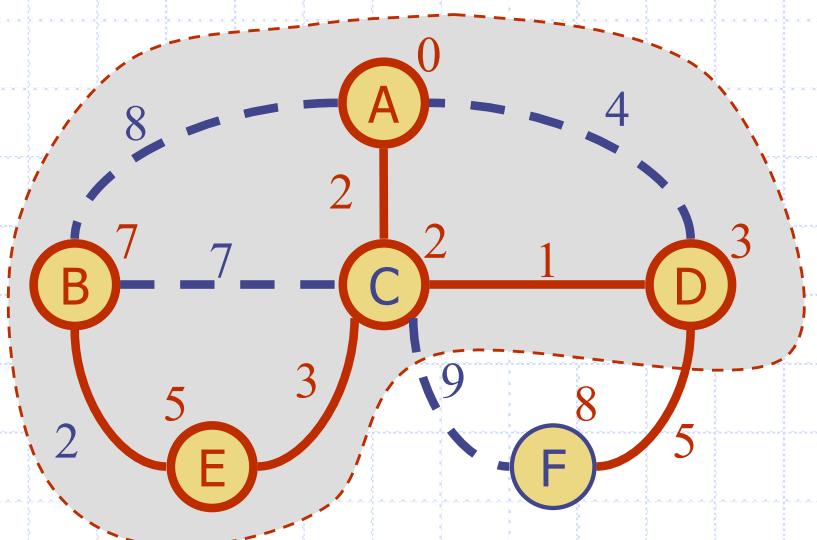


# Análisis del algoritmo

- Operaciones de Grafos
  - Encontramos todas las aristas incidentes una vez para cada vértice.
- Operaciones de etiquetas
  - Establecemos/obtenemos las etiquetas de distancia y localizador del vértice  $z$   $O(\deg(z))$  veces
  - Establecer/obtener una etiqueta lleva tiempo  $O(1)$
- Operaciones de una Cola de Prioridades
  - Cada vértice se inserta una vez y se elimina una vez de la cola de prioridad, donde cada inserción o eliminación toma tiempo  $O(\log n)$
  - La llave de un vértice en la cola de prioridad se modifica como máximo  $\deg(w)$  veces, donde cada cambio toma tiempo  $O(\log n)$
- El algoritmo de Dijkstra se ejecuta en tiempo  $O((n + m) \log n)$  siempre que el grafo esté representado por la estructura de lista de adyacencia
  - Recuerde que  $\sum_v \deg(v) = 2m$
- El tiempo de ejecución también se puede expresar como  $O(m \log n)$  ya que el grafo está conectado

# Cómo funciona el algoritmo?

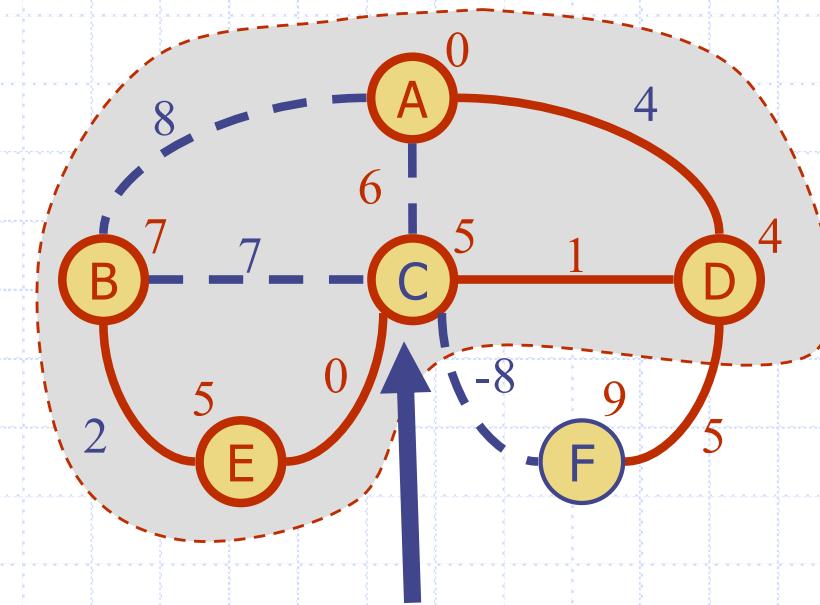
- El algoritmo de Dijkstra se basa en el método codicioso (greedy). Agrega vértices aumentando la distancia.
  - Supongamos que no encontró todas las distancias más cortas. Sea  $w$  el primer vértice incorrecto que procesó el algoritmo.
  - Cuando se consideró el nodo anterior,  $u$ , en el camino más corto verdadero, su distancia era correcta
  - Pero la arista  $(u, w)$  estaba **relajado** en ese momento!
  - Por lo tanto, siempre que  $D[w] \geq D[u]$ , la distancia de  $w$  no puede ser incorrecta. Es decir, no hay ningún vértice equivocado.



$(u, w) = (D, F)$  en este ejemplo

# Por qué no funciona para aristas con peso negativo

- ◆ El algoritmo de Dijkstra se basa en el método codicioso (greedy). Agrega vértices aumentando la distancia.
- Si un nodo con una arista incidente de peso negativo se agregara tarde a la nube, podría alterar las distancias de los vértices que ya están en la nube.



La distancia de C' es 1,  
pero ya está en la nube  
con  $d(C)=5$ !

# Algoritmo de Bellman-Ford

- Funciona incluso con aristas de peso negativo
- Debe asumir aristas dirigidas (de lo contrario tendríamos ciclos de peso negativo)
- La iteración  $i$  encuentra todos los caminos más cortos que usan  $i$  aristas.
- Tiempo de funcionamiento:  $O(nm)$ .
- Puede ampliarse para detectar un ciclo de peso negativo si existe.
  - ¿Cómo?

# Algoritmo Bellman-Ford: Detalles

**Algorithm** BellmanFordShortestPaths( $\vec{G}, v$ ):

**Input:** A weighted directed graph  $\vec{G}$  with  $n$  vertices, and a vertex  $v$  of  $\vec{G}$

**Output:** A label  $D[u]$ , for each vertex  $u$  of  $\vec{G}$ , such that  $D[u]$  is the distance from  $v$  to  $u$  in  $\vec{G}$ , or an indication that  $\vec{G}$  has a negative-weight cycle

$D[v] \leftarrow 0$

**for** each vertex  $u \neq v$  of  $\vec{G}$  **do**

$D[u] \leftarrow +\infty$

**for**  $i \leftarrow 1$  to  $n - 1$  **do**

**for** each (directed) edge  $(u, z)$  outgoing from  $u$  **do**

            // Perform the *relaxation* operation on  $(u, z)$

**if**  $D[u] + w((u, z)) < D[z]$  **then**

$D[z] \leftarrow D[u] + w((u, z))$

**if** there are no edges left with potential relaxation operations **then**

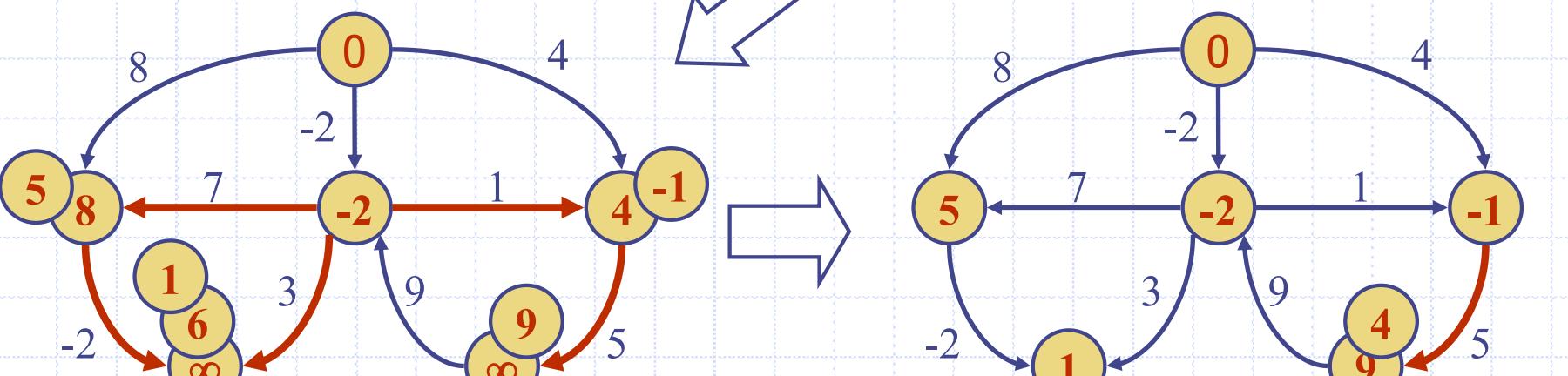
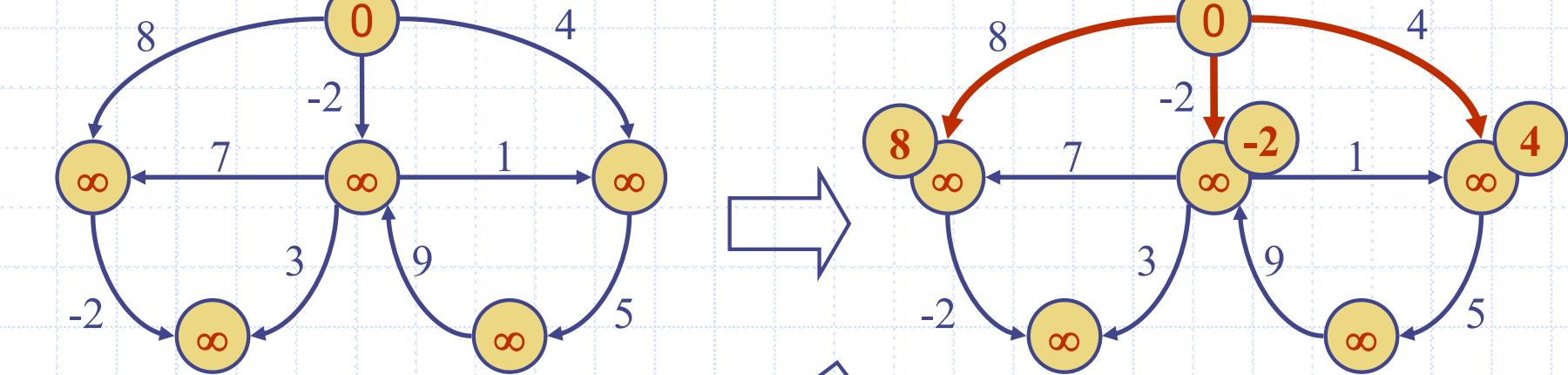
**return** the label  $D[u]$  of each vertex  $u$

**else**

**return** “ $\vec{G}$  contains a negative-weight cycle”

# Ejemplo de Bellman-Ford

Los nodos están etiquetados con su  $D[v]$  valores



# Algoritmo basado en DAG

- Podemos producir un algoritmo especializado de ruta más corta para grafos acíclicos dirigidos (DAG)
- Funciona incluso con aristas de peso negativo
- Utiliza orden topológico
- No utiliza estructuras de datos sofisticadas
- Es mucho más rápido que el algoritmo de Dijkstra.
- Tiempo de ejecución:  $O(n + m)$ .

# Algoritmo basado en DAG: detalles

**Algorithm**  $\text{DAGShortestPaths}(\vec{G}, s)$ :

**Input:** A weighted directed acyclic graph (DAG)  $\vec{G}$  with  $n$  vertices and  $m$  edges, and a distinguished vertex  $s$  in  $\vec{G}$

**Output:** A label  $D[u]$ , for each vertex  $u$  of  $\vec{G}$ , such that  $D[u]$  is the distance from  $v$  to  $u$  in  $\vec{G}$

Compute a topological ordering  $(v_1, v_2, \dots, v_n)$  for  $\vec{G}$

$D[s] \leftarrow 0$

**for** each vertex  $u \neq s$  of  $\vec{G}$  **do**

$D[u] \leftarrow +\infty$

**for**  $i \leftarrow 1$  to  $n - 1$  **do**

// Relax each outgoing edge from  $v_i$

**for** each edge  $(v_i, u)$  outgoing from  $v_i$  **do**

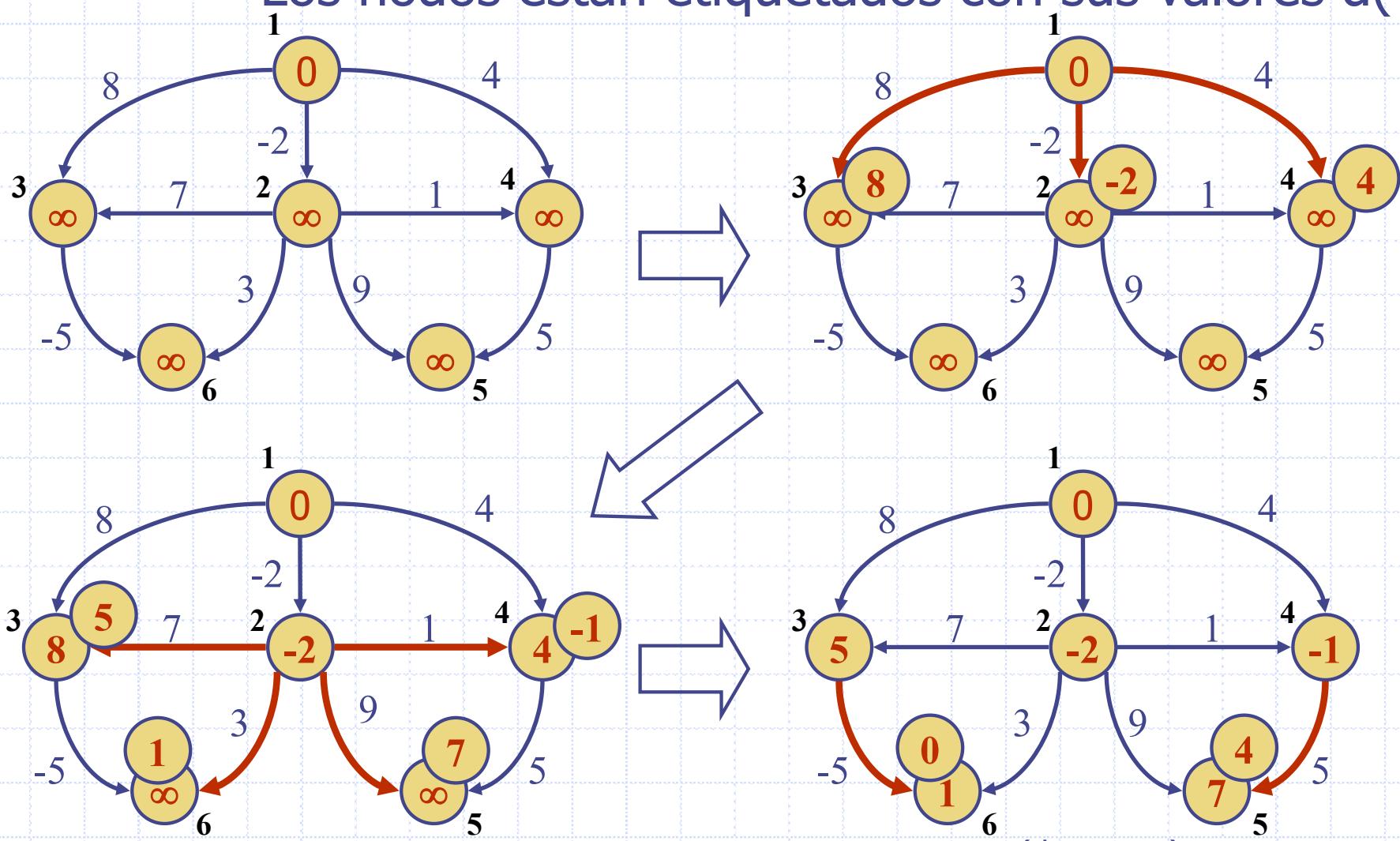
**if**  $D[v_i] + w((v_i, u)) < D[u]$  **then**

$D[u] \leftarrow D[v_i] + w((v_i, u))$

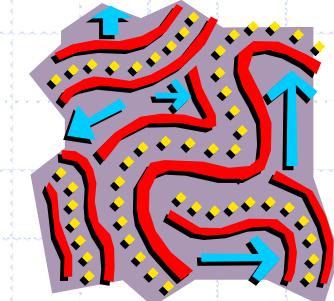
Output the distance labels  $D$  as the distances from  $s$ .

# Ejemplo de DAG

Los nodos están etiquetados con sus valores  $d(v)$



# Caminos más cortos para todos los pares



- Encuentre la distancia entre cada par de vértices en un grafo dirigido ponderado  $G$ .
- Podemos realizar  $n$  llamadas al algoritmo de Dijkstra (si no hay aristas negativas), lo que requiere tiempo  $O(n m \log n)$ .
- De la misma manera,  $n$  llamadas a Bellman-Ford tomarían tiempo  $O(n^2 m)$ .
- Podemos lograr un tiempo de  $O(n^3)$  usando programación dinámica (similar al algoritmo Floyd-Warshall ).

**Algoritmo *AllPair* (  $G$  )** {asume los vértices  $1, \dots, n$  }  
para todos *pares de vértices*  $(i,j)$

si  $i = j$

$D_0[i, i] \leftarrow 0$

else, si  $(i, j)$  es una arista en  $G$

$D_0[i, j] \leftarrow$  peso de la arista  $(i, j)$

else

$D_0[i, j] \leftarrow +\infty$

para  $k \leftarrow 1$  a  $n$  hacer

para  $yo \leftarrow 1$  a  $n$  hacer

para  $j \leftarrow 1$  a  $n$  hacer

$D_k[i, j] \leftarrow \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\}$

return  $D_n$

